

A prototype of a knowledge-based programming environment

Stef De Pooter, Johan Wittocx, and Marc Denecker

Department of Computer Science, K.U. Leuven

Abstract. In this paper we present a proposal for a knowledge-based programming environment. In such an environment, declarative background knowledge, procedures, and concrete data are represented in suitable languages and combined in a flexible manner. This leads to a highly declarative programming style. We illustrate our approach on an example and report about our prototype implementation.

1 Context

An obvious requirement for a powerful and flexible programming paradigm seems to be that within the paradigm different types of information can be expressed in suitable languages. However, most traditional programming paradigms and languages do not really have this property. In imperative languages, for example, non-executable background knowledge can not be described. The consequences become clear when we try to solve a scheduling problem in an imperative language: the background knowledge, the constraints that need to be satisfied by the schedule, gets mixed up with the algorithms. This makes adding new constraints and finding and modifying existing ones cumbersome.

On the other hand, most logic-based declarative programming paradigms lack the capacity to express procedures. Typically, they consist of a logic together with one specific type of inference. For example, Prolog uses Horn clause logic and does querying, in Description Logic the studied task is deduction, and Answer Set Programming and Constraint Programming make use of model generation. In such paradigms, whenever we try to perform a task that does not fit the inference mechanism at hand, the declarative aspect of the paradigm disappears. For example, when we try to solve a scheduling problem (which is a typical model-generation problem) in Prolog, then we need to represent the schedule as a term, say a list (rather than as a logical structure), and as a result the constraints do not really reside in the logic program, but will have to be expressed by clauses that iterate over a list [4]. Proving that a certain requirement is implied by another, is possible (in theory) for a theorem prover, but not in ASP. Etc.

To overcome these restrictions of existing paradigms, we propose a paradigm in which each component can be expressed in an appropriate language. We distinguish three components: procedures, (non-executable) background knowledge, and concrete data. For the first we need an imperative language, for the second an (expressive) logic, for the third a logical structure (which corresponds to a database). The connection between these components is mostly realized by various reasoning tasks, such as theorem proving, model generation, model checking, model revision, belief revision, constraint propagation, querying, datamining, visualization, etc.

The idea to support multiple forms of inference for the same logic or even for the same theories, was argued in [6]. Here it is argued that logic has a more flexible, multifunctional and therefore also more declarative role for problem solving than provided in many declarative programming paradigms, where typically one form of inference is central and theories are written to be used for this form of inference, sometimes even for a specific algorithm implementing this form of inference (such as PROLOG resolution). This view was therefore called the Knowledge Base System paradigm for declarative problem solving. The framework presented here is based on this view and goes beyond it in the sense that it offers a programming environment in which complex tasks can be programmed using multiple forms of inference and processing tools.

2 Overview of the language and system

To try out the above mentioned ideas in practice, we built a prototype interpreter that supports some basic reasoning tasks and a set of processing tools on high-level data such as vocabularies, structures and theories. In this section we will highlight various decisions in the design of our programming language and interpreter. In the next section we will illustrate the usage of the language with an example. We named our language DECLIMP, which is an aggregation of “declarative” and “imperative”.

2.1 Program structure

A DECLIMP program typically contains several blocks of code. Each block is either a procedure, a vocabulary (which is a list of sort, predicate and function names), a logic theory over vocabularies (which describes a piece of background knowledge using the relation and function names of its vocabulary), or a (possibly three-valued) structure over vocabularies. The latter represent databases over their vocabularies. To bring more structure into a program and to be able to work with multiple files, namespaces and include statements are provided.

Because vocabularies, logic theories and databases are not executable, and a program needs to be executed, control of a DECLIMP program is always in the hands of the procedures. Moreover, when a `main` procedure is available, the run of the program will start with the execution of this procedure. When there is no `main` procedure, the user can run commands in an interactive shell, after parsing the program.

In the next sections, we will describe the languages for the respective components in a DECLIMP program.

2.2 Knowledge representation language

For representing background knowledge we use an extended version of classical logic. A first advantage in using this language lies in the fact that classical logic is the best known and most studied logic. Also, classical logic has the important property that its informal semantics corresponds to its formal semantics. In other words, in classical logic the meaning of expressions¹ is intuitively clear. This is an important requirement in the design of a language that is accessible to a wider audience. Furthermore, there are already numerous declarative systems that use a language based on classical logic, or can easily be translated to it. Think of the languages of most theorem provers, various Description logics, and the language of model generators such as IDP [20,8] and ENFRAGMO [14].

Research in the Knowledge Representation and Reasoning community has clearly shown that classical logic is in many ways insufficient. Aggregates and (recursive) definitions are well-known concepts that are common in the background knowledge of many applications, and which can generally not, or not in a concise and intuitively clear manner, be expressed in first-order logic. In DECLIMP we use an order-sorted version of first-order logic, extended with inductive definitions [5], aggregates [15], (partial) functions and arithmetic.

2.3 Structures

Structures in DECLIMP are written in a simple language that allows to enumerate all elements that belong to a sort and all tuples that belong to a relation or function. As an alternative to enumerating a relation, it is also possible to specify the relation in a procedural way, namely as all the tuples for which a given procedure returns ‘true’. Furthermore, the interpretation of a function can be specified by a procedure, somewhat similar to “external procedure” in DLV [2].

As mentioned before, structures in DECLIMP are not necessarily two-valued. Three-valued structures are useful for representing incomplete information (which might be completed during the

¹ We mean expressions that occur in practice, not artificially constructed sentences that do not really have meaning in real life.

run of the program). To enumerate a three-valued relation (or function), two out of three of the following sets must be provided: tuples that certainly belong to the relation, tuples that certainly do not belong to the relation, and tuples for which it is unknown whether they belong to the relation or not. The third set can always be computed from the two given sets.

2.4 Procedures

The imperative programming language in our prototype system is LUA [9]. The main reason for this choice is the fact that LUA is a lightweight scripting language and also because it has a good C++ API [10]. This facilitates on the one hand the compilation of programs written in DECLIMP and, on the other hand, the integration with the components of our DECLIMP interpreter, which is written in C++. When we do not take those reasons into account, any other imperative language is candidate.

In procedures, various reasoning methods on theories and structures can be called. Currently, the most important tasks supported by the DECLIMP-interpreter are the following:

Finite model expansion: Given a three-valued structure S and a theory T , find a completion of S to a two-valued structure that satisfies T . This is essentially a generalization of the reasoning task performed by ASP solvers, constraint programming systems, Alloy analyzers, etc. It is suitable for problems such as scheduling, planning and diagnosis. In our DECLIMP interpreter, model expansion is implemented by calls to the IDP system [20], which consists of the grounder GIDL [21] and solver MINISATID [11].

Finite model checking: Check whether a given two-valued structure is a model of a theory. This is an instance of model expansion and is implemented as such.

Constraint propagation: Deduce facts that must hold in all models of a given theory which complete a given three-valued structure. This is a useful mechanism in configuration systems [18] and for query answering in incomplete databases [3]. The propagation algorithm we implemented is described in [19].

Querying: Given an FO formula φ and a two-valued structure S , find all substitutions for free variables of φ that make φ true in S . The implementation of this mechanism makes use of Binary Decision Diagrams as described in [21].

Theorem proving: Given two FO theories T_1 and T_2 , check whether $T_1 \models T_2$. This is implemented by calling a theorem prover provided by the user. In principle, any theorem prover that accepts TPTP [16] can be used.

Visualization: Show a visual representation of a given structure. We implement this by calling IDPDRAW, a tool for visualizing finite structures in which visual output is specified declaratively by definitions in our knowledge representation language or in ASP.

The values returned by the reasoning methods can be used in other reasoning methods and LUA-statements. We will illustrate this with an example in the next section.

3 Programming in DECLIMP

Say we want to write an application that allows players to solve sudoku puzzles. Such an application should be able to perform tasks such as generating puzzles, showing puzzles on the screen, checking whether solutions (player's choices) satisfy the sudoku rules, giving hints to the player, etc. In this application the different components we described before can clearly be distinguished: (1) the background knowledge consists of a logic theory containing the well-known sudoku constraints;

$$\begin{aligned}
&\forall r \forall n \exists! c : \text{Sudoku}(r, c) = n \\
&\forall c \forall n \exists! r : \text{Sudoku}(r, c) = n \\
&\forall b \forall n \exists! r \exists! c : \text{InBlock}(b, r, c) \wedge \text{Sudoku}(r, c) = n \\
&\forall b \forall r \forall c : \text{InBlock}(b, r, c) \Leftrightarrow b = ((r - 1)/3) * 3 + ((c - 1)/3) + 1
\end{aligned}$$

(2) the data is stored in logical structures representing puzzles, and (partial and complete) solutions; and (3) the tasks we want it to perform, can be implemented using well-known inference methods.

Below we show (part of) a DECLIMP program. This code shows the use of an include statement and a namespace, and the declaration of a vocabulary `sudokuVoc` and a theory `sudokuTheory`, where the latter is simply an ASCII version of the theory shown above. Also note the `main` procedure at the bottom, which will be called when this program is passed to the interpreter.

```
#include "grid.idp"

namespace sudoku {

  vocabulary sudokuVoc {
    extern vocabulary grid::simpleGridVoc
    type Num isa nat
    type Block isa nat
    Sudoku(Row,Col) : Num
    InBlock(Block,Row,Col)
  }

  theory sudokuTheory : sudokuVoc {
    ! r n : ?1 c : Sudoku(r,c) = n.
    ! c n : ?1 r : Sudoku(r,c) = n.
    ! b n : ?1 r c : InBlock(b,r,c) & Sudoku(r,c) = n.
    ! r c b : InBlock(b,r,c) <=> b = ((r-1)/3)*3 + ((c-1)/3) + 1.
  }

  procedure solve(input) {
    return modelExpand(sudokuTheory,input)
  }

  procedure printSudoku(puzzle) {
    -- code for visualizing a sudoku puzzle.
  }

  procedure createSudoku() {
    math.randomseed(os.time())
    local puzzle = grid::makeEmptyGrid(9) -- defined in grid.idp

    stdoptions.nrmodels = 2
    local currSols = modelExpand(sudokuTheory,puzzle)
    while #currSols > 1 do
      repeat
        col = math.random(1,9)
        row = math.random(1,9)
        num = currSols[1][sudokuVoc::Sudoku](row,col)
      until num ~= currSols[2][sudokuVoc::Sudoku](row,col)
      makeTrue(puzzle[sudokuVoc::Sudoku].graph,{row,col,num})
      currSols = modelExpand(sudokuTheory,puzzle)
    end

    printSudoku(puzzle)
  }
}

procedure main() {
  sudoku::createSudoku()
}
```

Let us have a closer look at procedure `createSudoku` for creating sudoku puzzles. First it initializes an empty puzzle by instantiating a new logical structure. This is done by calling a procedure `makeSquareGrid` which instantiates a structure with data about a generic grid of a certain size, and then adding domains for numbers and blocks particular for sudoku grids.

The second part of the procedure adds numbers to the grid until there is only one solution left for the puzzle. This is realized by performing model expansion (by calling `modelExpand`) to find two models of the theory that extend the given partially filled in puzzle. When two models are found, the algorithm selects a number that is unique for the first solution (that is, the number at the same position in the second solution is different) and is not yet present in the puzzle. When such an entry is found, it is added to the puzzle by making the tuple `{row,col,num}` true in the interpretation of the function `Sudoku(Row,Col):Num`. Next, the procedure ask for two new models, and the process starts over. When only one model is found, the iteration stops, and procedure `printSudoku` is called to show the result on the screen using the visualization tool mentioned in the previous section.

4 Related work

There have been many proposals in the literature to combine procedural and declarative languages. A frequently occurring combination is that of a procedural language in which a program can post constraints expressed in an (often ad-hoc) declarative constraint language, while other primitives allow to call the constraint-solving process on the constraint store, express heuristics or call other processes, for example to edit or visualize output. Examples of systems with such languages are CPLEX [1], MOZART [17] and COMET [13]. These systems differ from DECLIMP in the sense that they offer only one kind of inference, namely constraint solving. A similar remark can be made about CLP and PROLOG systems with support for constraint propagation. Here the “procedural language” is the PROLOG language under its procedural semantics. In our system high-level concepts such as vocabularies, theories and structures are treated as first-class citizens that can be operated upon by arbitrary inference and processing tools, which offers more flexibility.

For another group of systems, control over execution of programs is in hands of one inference mechanism – or at least that inference is the main mechanism – and an integrated procedural language then allows users to steal some aspects of the inference mechanism, or for example format input and output, but do not allow to take over control. Examples of such systems are CLINGO [7] and ZINC [12]. The procedural languages in these systems have a more limited task then the one in DECLIMP. In DECLIMP the procedures are in control during execution, not just one of the inference mechanisms.

5 Conclusion

We have presented a knowledge-based programming environment, providing a declarative language for expressing background knowledge, an imperative programming language for writing procedures, and logical structures for expressing concrete data. The system also provides some state-of-the-art inference tools for performing various reasoning tasks.

We believe that a programming environment like the one proposed here overcomes some of the limitations of “single-programming-style” paradigms, by allowing a programmer to express the different types of information in software applications in appropriate languages. Making this explicit distinction in different types of information will increase readability, maintainability and reusability of programming code.

References

1. IBM ILOG CPLEX optimizer. <http://www.ibm.com/software/integration/optimization/cplex-optimizer>.

2. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: Theory and implementation. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 407–424. Springer Berlin / Heidelberg, 2008.
3. Marc Denecker, Álvaro Cortés Calabuig, Maurice Bruynooghe, and Ofer Arieli. Towards a logical reconstruction of a theory for locally closed databases. *ACM Transactions on Database Systems*, 35(3):22:1–22:60, 2010.
4. Marc Denecker, Danny De Schreye, and Yves Willems. Terms in Logic programs: a problem with their semantics and its effect on the programming methodology. *CCAI: Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, 7(3-4):363–383, 1990.
5. Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 9(2):Article 14, 2008.
6. Marc Denecker and Joost Vennekens. Building a knowledge base system for an integration of logic programming and classical logic. In Maria Garcia de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *LNCS*, pages 71–76. Springer, 2008.
7. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user’s guide to gringo, clasp, clingo, and iclingo. <http://downloads.sourceforge.net/potassco/guide.pdf>, 2010.
8. The IDP system. <http://dtai.cs.kuleuven.be/krr/software.html>.
9. Roberto Ierusalimsky, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua – an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
10. Roberto Ierusalimsky, Luiz Henrique de Figueiredo, and Waldemar Celes. Passing a language through the eye of a needle. *Queue*, 9:20:20–20:29, May 2011.
11. Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *LNCS*, pages 211–224. Springer, 2008.
12. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
13. Laurent Michel and Pascal Van Hentenryck. The Comet programming language and system. In Peter van Beek, editor, *CP*, volume 3709 of *LNCS*, pages 881–881. Springer, 2005.
14. David G. Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebbi. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, Simon Fraser University, Canada, 2006.
15. Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming (TPLP)*, 7(3):301–353, 2007.
16. Geoff Sutcliffe. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
17. Peter Van Roy, editor. *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.
18. Hanne Vlaeminck, Joost Vennekens, and Marc Denecker. A logical framework for configuration software. In António Porto and Francisco Javier López-Fraguas, editors, *PPDP*, pages 141–148. ACM, 2009.
19. Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. Constraint propagation for extended first-order logic. *CoRR*, abs/1008.2121, 2010.
20. Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In Marc Denecker, editor, *LaSh*, pages 153–165, 2008.
21. Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, 38:223–269, 2010.